JÜLICH
FORSCHUNGSZENTRUM

Parallel I/O and Portable Data Formats

Parallel netCDF

16 March 2015 | Sebastian Lührs

Outline

- Introduction
- Basic file handling
- Advanced file operations
- Exercises

Introduction to Parallel netCDF

- netCDF is a portable, self-describing file format developed by Unidata at UCAR (University Cooperation for Atmospheric Research)
- netCDF does not provide a parallel API prior to 4.0
  - Classic and 64-bit offset file format
- pnetCDF is maintained by Argonne National Laboratory
  - `http://trac.mcs.anl.gov/projects/parallel-netcdf/`

Terms and definitions

JÜLICH FORSCHUNGSZENTRUM

Dimension

An entity that can either describe a physical dimension of a dataset, such as time, latitude, etc., as well as index to sets of stations or model-runs.

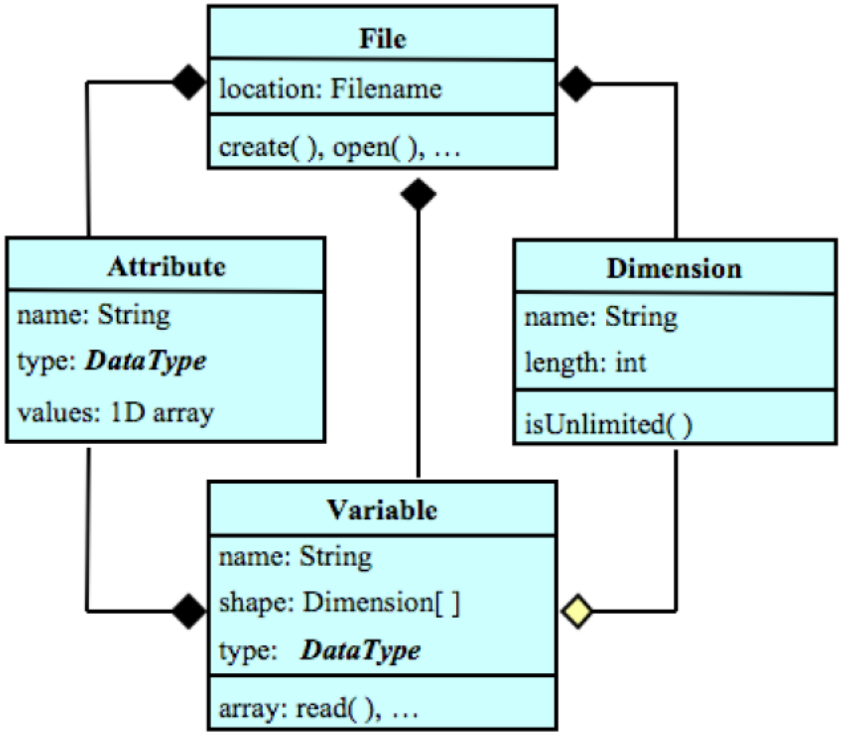
Variable

An entity that stores the bulk of the data. It represents an n-dimensional array of values of the same type.

Attribute

An entity to store data on the datasets contained in the file or the file itself. The latter are called *global attributes*.

NetCDF Classic model



Variables and attributes have one of six primitive data types.

DataType
char
byte
short
int
float
double

A file has named variables, dimensions, and attributes. Variables also have attributes. Variables may share dimensions, indicating a common grid. One dimension may be of unlimited length.

source: Hartnett, E., 2010-09: NetCDF and HDF5 - HDF5 Workshop 2010.

Naming conventions

Dimensions, variables, attributes

- Sequence of alphanumeric characters, `_`, `.`, `+`, `-`, `@`
- Must begin with a letter or underscore
  - Name with underscores are reserved for system use
- Names are case sensitive
- Other conventions may restrict names even more

Dimensions

- Can represent a physical dimension like time, height, latitude, longitude, etc.
- Can be used to index other quantities, e.g., station number
- Have a name and length
- Can have either a fixed length or ’UNLIMITED’
  - In *classic* and *64bit offset* files at most one
- Used to define the shape of variables
- Can be used more than once in a variable declaration
  - Use only more than once, where semantically useful

Variables

- Store the bulk data in the dataset
- Regarded as *n*-dimensional array
  - Scalar values represented as 0-dimensional arrays
- Have a name, type and shape
  - Shape is defined through dimensions
- Once created, cannot be deleted or altered in shape
- Variable type must be one of the basic types
  - byte, character, short, int, float, double
- Variables with one unlimited dimension are called *record variables*, otherwise *fixed variables*
- A position along a dimension can be specified as index
  - Starting at 0 in C and 1 in Fortran

Coordinate variables

- Variables can have the same name as dimensions
- Have no special semantic in netCDF itself
- By convention, applications using netCDF should treat them in a special way
- Usually describes a coordinate corresponding to that dimension
- Each coordinate variable is a vector that's shape is defined by the dimension of the same name
- Might provide a more convenient way to access the data
    - By convention, current applications assume coordinate variables to be numeric and strictly monotonic

Attributes

- Used to store meta data of variables or the complete data set (global attributes)
- Have a name, a type, a length, and a value
- Treated as vector
  - Scalar values a single-element vectors
- Can be deleted and changed in shape at any time
- Please adhere existing conventions for attributes

Attribute Conventions

`units` – character string that specifies the units used for a variable

`long_name` – long descriptive name for a variable

`valid_min` – value specifying the minimum valid value for a variable

`valid_max` – value specifying the maximum valid value for a variable

`valid_range` – vector of two numbers specifying the minimum and maximum valid value for a variable

…

For more, please read the Appendix B: Attribute Conventions of the netCDF User Guide

Datatypes

The netCDF classic and 64-bit offset file format only support basic types

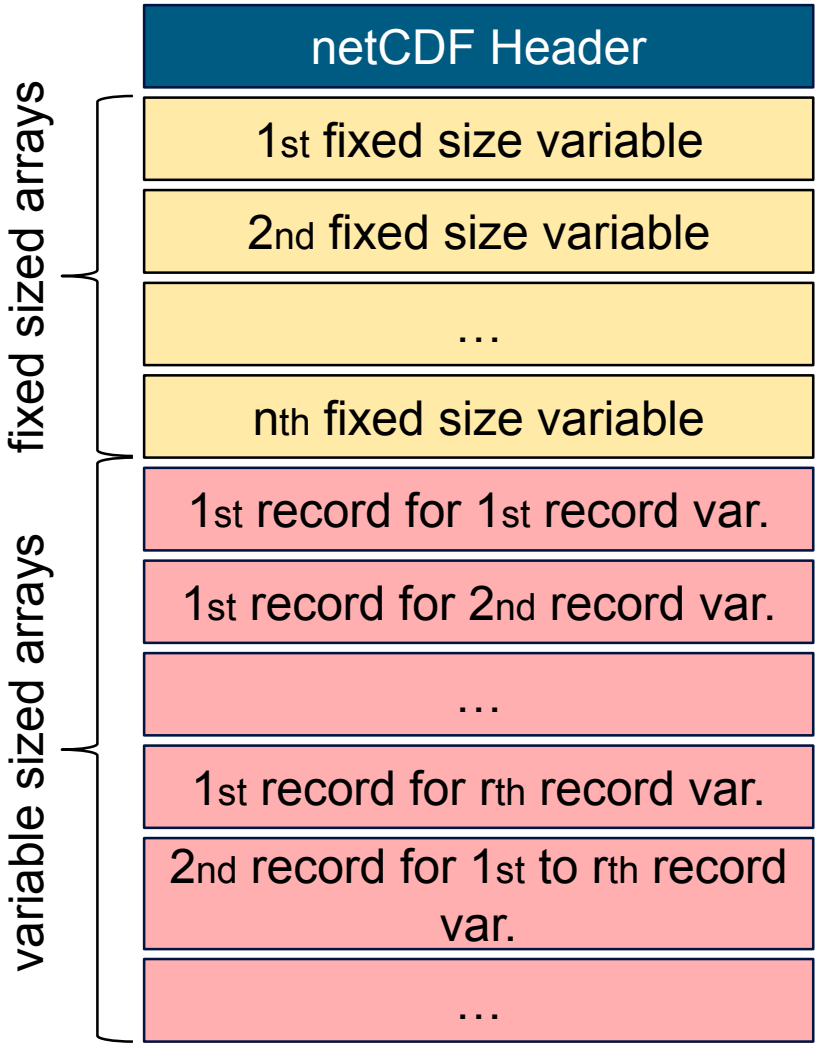
C	Fortran	Storage
NC_BYTE	nf_byte	8-bit signed integer
NC_CHAR	nf_char	8-bit unsigned integer
NC_SHORT	nf_short	16-bit signed integer
NC_INT	nf_int	32-bit signed integer
NC_FLOAT	nf_float	32-bit floating point
NC_DOUBLE	nf_double	64-bit floating point

The netCDF file format

netCDF dataset definition

n arrays of fixed dimensions

r arrays with its most significant dimension set to UNLIMITED records are defined by the remaining dimensions



netCDF file format characteristics

- A netCDF (classic and 64-bit offset format) file consists of three regions
  - Header
  - Non-record variables, multi-dimensional data with fixed size in each dimension
  - Record variables, multi-dimensional data with a single dimension of UNLIMITED size, and the remaining dimensions fixed
- All data is written in big-endian format in an internal format similar to XDR

Performance Implications

- The header is dense, i.e., changing the header after variables have been added, will result in the copy of all subsequent data
  - Avoid later additions and renaming of netCDF components
  - Use `nc_enddef` to reserve header space
- Record variables are interleaved
  - Using more than one per file will result in non-contiguous buffers, and performance degradation is likely

netCDF classic format limitations

- If no unlimited dimension is used, only one variable can exceed 2 GiB (but it can be as large as the FS permits)
  - It must be the last variable in the data set
  - The start offset must be less than $2^{31} - 4$ bytes (approx. 2 GiB)
- If the unlimited dimension is used, record variables may exceed 2 GiB in size
  - The start offset of each record variable within each record must be less than $2^{31} - 4$ bytes (approx. 2 GiB)

netCDF 64-bit offset format limitations

- If no unlimited dimension is used, only one variable can exceed 4 GiB (but it can be as large as the FS permits)
  - It must be the last variable in the data set
- A data set can contain $2^{32} - 1$ fixed sized variables, each less 4 GiB in size
- A record variable cannot use more than 4 GiB of storage for each record's worth of data
  - maximum: $2^{32} - 1$ records, of up to $2^{32} - 1$ variables

Outline

- Introduction
- Basic file handling
- Advanced file operations
- Exercises

Workflow: Creating a netCDF data set

- Create a new dataset
  - A new file is created and netCDF is left in define mode
- Describe contents of the file
  - Define dimensions for the variables
  - Define variables using the dimensions
  - Store attributes if needed
- Switch to data mode
  - Header is written and definition of the file content is completed
- Store variables in file
  - Parallel netCDF distinguishes between collective and individual data mode
  - Initially in collective mode, user has to switch to individual data mode explicitely
- Close file

Header files

C
```
#include <pnetcdf.h>
```

Fortran
```
use pnetcdf

include 'pnetcdf.inc'
```

- Contain definition of
  - constants
  - functions

Creating a file

C

```
int ncmpi_create(MPI_Comm comm,
                 const char* filename, int cmode,
                 MPI_Info info, int* ncid)
```

Fortran

```
INTEGER NFMPI_CREATE(COMM, FILENAME, CMODE, INFO,
                     NCID)
  CHARACTER*(*) FILENAME
  INTEGER COMM, MODE, INFO, NCID
```

- Call is collective over `comm`
- `ncid` is the id of the internal file handle
- `cmode` must specify at least one of the following
  - `(NC/NF)_CLOBBER` – Create new file and overwrite, if it existed before
  - `(NC/NF)_NOCLOBBER` – Create new file only, if it did not exist before
- Choose file format on file creation
  - 32-bit offsets (default)
  - `(NC/NF)_64BIT_OFFSET` – 64-bit offsets

Open an existing netCDF data set

C

```
int ncmpi_open(MPI_Comm comm, const char* filename,
               int omode, MPI_Info info, int* ncid)
```

Fortran

```
INTEGER NFMPI_OPEN(COMM, FILENAME, OMODE, INFO,
                   NCID)
  CHARACTER*(*) FILENAME
  INTEGER COMM, OMODE, INFO, NCID
```

- Call is collective over `comm`
- `ncid` is the id of the internal file handle
- `omode` must specify at least one of the following
  - `(NC/NF)_WRITE` – Open file for any kind of change to the file
  - `(NC/NF)_NOWRITE` – Open the file read-only

Closing a file

C
```
int ncmpi_close(int ncid)
```

Fortran
```
INTEGER NFMPI_CLOSE(NCID)
  INTEGER NCID
```

- Close file associated with `ncid`

Exercise

Exercise 1 – NetCDF hello world

- Create a generic parallel application (C, Fortran) which creates an empty netcdf file
- Compile, link and execute the application

```
# Compile, Link Juropa
module load parallel-netcdf
mpicc -I$PNETCDF_INCLUDE helloworld.c
  -L$PNETCDF_LIB -lpnetcdf -o helloworld

mpif90 -I$PNETCDF_INCLUDE helloworld_f.F90
  -L$PNETCDF_LIB -lpnetcdf -o helloworld

# start interactive computing session:
msub -I -X -l nodes=1:ppn=8,walltime=06:00:00

# execute program
mpiexec -np 8 helloworld
```

Defining dimensions

C

```
int ncmpi_def_dim(int ncid, const char* name,
                  MPI_Offset len, int* dimid)
```

Fortran

```
INTEGER NFMPI_DEF_DIM(NCID, NAME, LEN, DIMID)
  CHARACTER*(*) NAME
  INTEGER NCID, DIMID
  INTEGER(KIND=MPI_OFFSET_KIND) LEN
```

- `name` represents the name of the dimension
- `len` represents the value
  - `(NC/NF)_UNLIMITED` will create an unlimited dimension
- Can only be called in *definition mode*

Defining variables

C

```
int ncmpi_def_var(int ncid, const char* name,
                  nc_type xtype, int ndims,
                  const int* dimids, int* varid)
```

Fortran

```
INTEGER NFMPI_DEF_VAR(NCID, NAME, XTYPE, NDIMS,
                      DIMIDS, VARID)
  CHARACTER*(*) NAME
  INTEGER, NCID, XTYPE, NDIMS, VARID
  INTEGER(*) DIMIDS
```

- `xtype` specifies the external type of this variable
- `dimids` is an array of size `ndims`

Defining attributes

C

```
int ncmpi_put_att_<type>(int ncid, int varid,
                const char* name, nc_type xtype,
                MPI_Offset len, const <type>*attr)
```

Fortran

```
INTEGER NFMPI_PUT_ATT_<type>(NCID, VARID, NAME,
                                XTYPE, LEN, ATTR)
  <type> ATTR
  CHARACTER*(*) NAME
  INTEGER NCID, VARID, XTYPE
  INTEGER(KIND=MPI_OFFSET_KIND) LEN
```

- Puts the attribute `attr` into the data set
- `varid` is the id annotated variable, or 0, if it is a global attribute
- `xtype` specifies the external type of this attribute

Reading attributes

C

```
int ncmpi_get_att_<type>(int ncid, int varid,
                         const char* name, <type>*attr)
```

Fortran

```
INTEGER NFMPI_GET_ATT_<type>(NCID, VARID, NAME,
                             ATTR)
  <type> ATTR
  CHARACTER*(*) NAME
  INTEGER NCID, VARID
```

- Reads the attribute `attr` from the data set
- `varid` is the id annotated variable, or 0 `(NC/NF)_GLOBAL`), if it is a global attribute

Closing define mode

C
```
int ncmpi_enddef(int ncid)
```

Fortran
```
INTEGER NFMPI_ENDDEF(NCID)
  INTEGER NCID
```

- Ends the definition phase, and switches to collective data mode
- Once variables have been put into the data set, definitions should not be altered

Writing variables collectively

C

```
int ncmpi_put_vara_<type>_all(int ncid, int varid,
                      const MPI_Offset start[],
                      const MPI_Offset count[],
                      const <type>* var)
```

Fortran

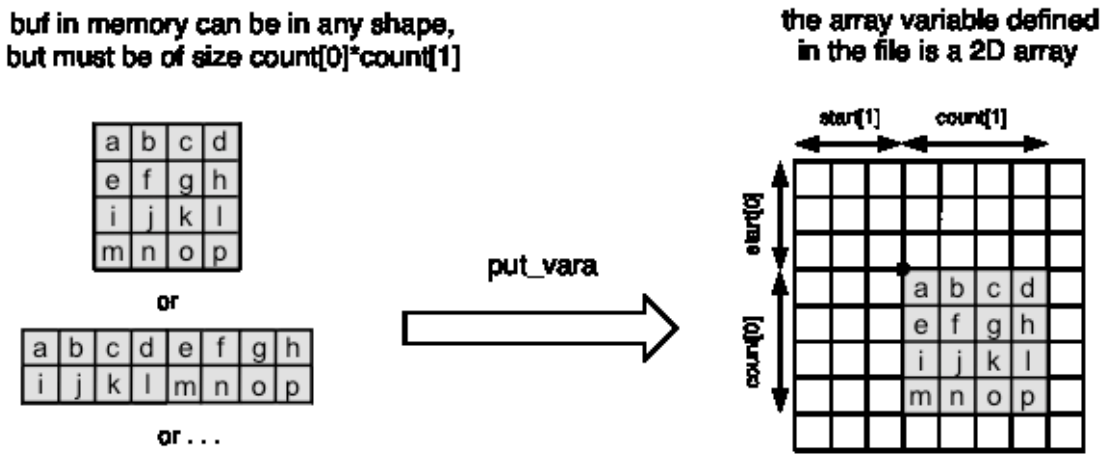
```
INTEGER NFMPI_PUT_VARA_<type>_ALL(NCID, VARID,
                                  START, COUNT, VAR)
  <type>(*) VAR
  INTEGER NCID, VARID
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT
```

- Writes a *slab* of data to the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in collective data mode

Writing variables collectively

C

```
int ncmpi_put_vara_<type>_all(int ncid, int varid,
                              const MPI_Offset start[],
                              const MPI_Offset count[],
                              const <type>* var)
```



source: PnetCDF C Interface Guide, http://cucis.ece.northwestern.edu/projects/PnetCDF/doc/pnetcdf-c/

Writing variables individually

C

```
int ncmpi_put_vara_<type>(int ncid, int varid,
                          const MPI_Offset start[],
                          const MPI_Offset count[],
                          const <type>* var)
```

Fortran

```
INTEGER NFMPI_PUT_VARA_<type>(NCID, VARID, START,
                              COUNT, VAR)
  <type>(*) VAR
  INTEGER NCID, VARID
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT
```

- Writes a *slab* of data to the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in individual data mode

Reading variables collectively

C

```
int ncmpi_get_vara_<type>_all(int ncid, int varid,
                      const MPI_Offset start[],
                      const MPI_Offset count[],
                      <type>* var)
```

Fortran

```
INTEGER NFMPI_GET_VARA_<type>_ALL(NCID, VARID,
                                  START, COUNT, VAR)
  <type>(*) VAR
  INTEGER NCID, VARID
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT
```

- Writes a *slab* of data to the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in collective data mode

Reading variables individually

C

```
int ncmpi_get_vara_<type>(int ncid, int varid,
                          const MPI_Offset start[],
                          const MPI_Offset count[],
                          <type>* var)
```

Fortran

```
INTEGER NFMPI_GET_VARA_<type>(NCID, VARID, START,
                              COUNT, VAR)
  <type>(*) VAR
  INTEGER NCID, VARID
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT
```

- Writes a *slab* of data to the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in individual data mode

Switching data modes

C
```
int ncmpid_begin_indep_data(int ncid)
```

Fortran
```
INTEGER NFMPI_BEGIN_INDEP_DATA(NCID)
  INTEGER NCID
```

- Switches from collective data mode to individual data mode

C
```
int ncmpid_end_indep_data(int ncid)
```

Fortran
```
INTEGER NFMPI_END_INDEP_DATA(NCID)
  INTEGER NCID
```

- Switches from individual data mode to collective data mode

Example Write (C), Part I

```
/* from pnetcdf tutorial: simple demonstration of pnetcdf text
   attribute on dataset write out rank into 1-d array collectively.
   The most basic way to do parallel i/o with pnetcdf */
#include <stdlib.h>
#include <mpi.h>
#include <pnetcdf.h>
#include <stdio.h>
static void handle_error(int status){
    fprintf(stderr, "%s", ncmpi_strerror(status));exit(-1);}
int main(int argc, char **argv) {
    int ret, ncfile, nprocs, rank, dimid, varid1, varid2, ndims=1;
    MPI_Offset start, count=1; int data;
    char buf[13] = "Hello World";

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    ret = ncmpi_create(MPI_COMM_WORLD, argv[1],
                       NC_CLOBBER|NC_64BIT_OFFSET,
                       MPI_INFO_NULL, &ncfile);
    if (ret != NC_NOERR) handle_error(ret);
    ret = ncmpi_def_dim(ncfile, "d1", nprocs, &dimid);
    if (ret != NC_NOERR) handle_error(ret);
```

Example Write (C), Part II

```
    ret = ncmpi_def_var(ncfile, "v1", NC_INT, ndims, &dimid, &varid1);
    if (ret != NC_NOERR) handle_error(ret);
    ret = ncmpi_def_var(ncfile, "v2", NC_INT, ndims, &dimid, &varid2);
    if (ret != NC_NOERR) handle_error(ret);

    ret = ncmpi_put_att_text(ncfile, NC_GLOBAL, "string", 13, buf);
    if (ret != NC_NOERR) handle_error(ret);

    ret = ncmpi_enddef(ncfile); if (ret != NC_NOERR) handle_error(ret);
    start=rank, count=1, data=rank;

    /* in this simple example every process writes its rank to
       two 1d variables */
    ret = ncmpi_put_vara_int_all(ncfile, varid1, &start, &count, &data);
    if (ret != NC_NOERR) handle_error(ret);

    ret = ncmpi_put_vara_int_all(ncfile, varid2, &start, &count, &data);
    if (ret != NC_NOERR) handle_error(ret);
    ret = ncmpi_close(ncfile);
    if (ret != NC_NOERR) handle_error(ret);
    MPI_Finalize();
    return 0;
}
```

Motivation for data set inquiry

- A generic application should be able to handle the data set correctly
- Semantic information must be encoded in names and attributes
  - Conventions need to be set up and used for a given data set class
- Data set structure can be reconstructed from the file

Workflow: Reading a netCDF data set

- Open a data set
- Inquire contents of data set
  - Inquire dimensions for allocation dimensions
  - Inquire variables for id of the desired variable
  - Inquire attributes for additional information
- Allocate memory according to shape of variables
- Read variables from file
  - Parallel netCDF distinguishes between collective and individual data mode
  - Initially in collective mode, user has to switch to individual data mode explicitly
- Close file

Inquiry of number of data set entities

C
```
int ncmpi_inq_ndims(int ncid,  int* ndims)
```

Fortran
```
INTEGER NFMPI_INQ_NDIMS(NCID, NDIMS)
  INTEGER NCID, NDIMS
```

- Query number of dimensions

C
```
int ncmpi_inq_nvars(int ncid, int* nvars)
```

Fortran
```
INTEGER NFMPI_INQ_NVARS(NCID, NVARS)
  INTEGER NCID, NVARS
```

- Query number of variables

C
```
int ncmpi_inq_natts(int ncid, int* natts)
```

Fortran
```
INTEGER NFMPI_INQ_NATTS(NCID, NATTS)
  INTEGER NCID, NATTS
```

- Query number of attributes

Inquiry of ids of data set entities

JÜLICH FORSCHUNGSZENTRUM

C

```
int ncmpi_inq_varid(int ncid, const char* name,
                    int* varid)
```

Fortran

```
INTEGER NFMPI_INQ_VARID(NCID, NAME, VARID)
  INTEGER NCID, VARID
  CHARACTER*(*) NAME
```

- Query id of variable given by `name`

C

```
int ncmpi_inq_vardimid(int ncid, int varid,
                       int* dimids)
```

Fortran

```
INTEGER NFMPI_INQ_VARDIMID(NCID, VARID, DIMIDS)
  INTEGER NCID, VARID,
  INTEGER* DIMIDS
```

- Query `dimids` for given `varid`

Inquiry of dimension lens

C

```
int ncmpi_inq_dimlen(int ncid, int dimid,
                     MPI_Offset* len)
```

Fortran

```
INTEGER NFMPI_INQ_DIMLEN(NCID, DIMID, LEN)
  INTEGER NCID, DIMID
  INTEGER(KIND=MPI_OFFSET_KIND) LEN
```

- reads `len` from a given dimension

Exercise

Exercise 2 – 1d array

1.1 Write

- Create a NetCDF data set, containing one one-dimensional variable
- A local vector of 10000 integers should be allocated and initialized with the task number
- Each task should write the vector to the NetCDF data set as a part of the global vector
- Write should be collective

1.2 Read

- Read the NetCDF data set into memory
- Each task should first read in the dimension and size of the NetCDF variable
- Allocate then the memory for the local vector
- Read should be collective
- Check if the data is consistent (task number)

Outline

- Introduction
- Basic file handling
- Advanced file operations
  - Flexible data mode interface
  - Data set inquiry
  - Release Info
- Exercises

Motivation for flexible API

- Original interface brings a lot of function name cruft with individual function calls for each data type
- The flexible data mode API is used in the background
- The user can also specify the in-memory storage using MPI data types

Writing variables collectively

C

```
int ncmpi_put_vara_all(int ncid, int varid,
          const MPI_Offset start[],
          const MPI_Offset count[],
          const void* buf, const MPI_Offset elements,
          MPI_Datatype datatype)
```

Fortran

```
INTEGER NFMPI_PUT_VARA_ALL(NCID, VARID, START,
                           COUNT, BUF,
                           ELEMENTS, DATATYPE)
  <type>(*) BUF
  INTEGER NCID, VARID, DATATYPE
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT,
          ELEMENTS
```

- Writes a *slab* of data to the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in collective data mode
- `start`, `count` refer to the data in the file
- `buf`, `elements`, and `datatype` refer to the data in memory

Reading variables collectively

C

```
int ncmpi_get_vara_all(int ncid, int varid,
          const MPI_Offset start[],
          const MPI_Offset count[],
          const void* buf, const MPI_Offset elements,
          MPI_Datatype datatype)
```

Fortran

```
INTEGER NFMPI_GET_VARA_ALL(NCID, VARID, START,
                           COUNT, BUF,
                           ELEMENTS, DATATYPE)
  <type>(*) BUF
  INTEGER NCID, VARID, DATATYPE
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT,
          ELEMENTS
```

- Reads a *slab* of data to the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in collective data mode
- `start`, `count` refer to the data in the file
- `buf`, `elements`, and `datatype` refer to the data in memory

Non-blocking interface

- Can aggregate multiple smaller requests into larger ones for better I/O performance

C

```
int ncmpi_iput_vara_<type>(int ncid, int varid,
                           const MPI_Offset start[],
                           const MPI_Offset count[],
                           const <type>* var,
                           int* req_id)
```

Fortran

```
INTEGER NFMPI_IPUT_VARA_<type>(NCID, VARID, START,
                               COUNT, VAR, REQ_ID)
  <type>(*) VAR
  INTEGER NCID, VARID, REQ_ID
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT
```

- `req_id` must be stored for later access

Non-blocking interface

C

```
int ncmpi_wait(int ncid, int num_reqs,
               int req_ids[], int statuses[])
int ncmpi_wait_all(int ncid, int num_reqs,
                   int req_ids[], int statuses[])
```

Fortran

```
INTEGER NFMPI_WAIT(NCID, NUM_REQS, REQ_IDS,
                   STATUSES)
  INTEGER NCID, NUM_REQS,
  INTEGER(*) REQ_IDS, STATUSES
INTEGER NFMPI_WAIT_ALL(NCID, NUM_REQS, REQ_IDS,
                       STATUSES)
```

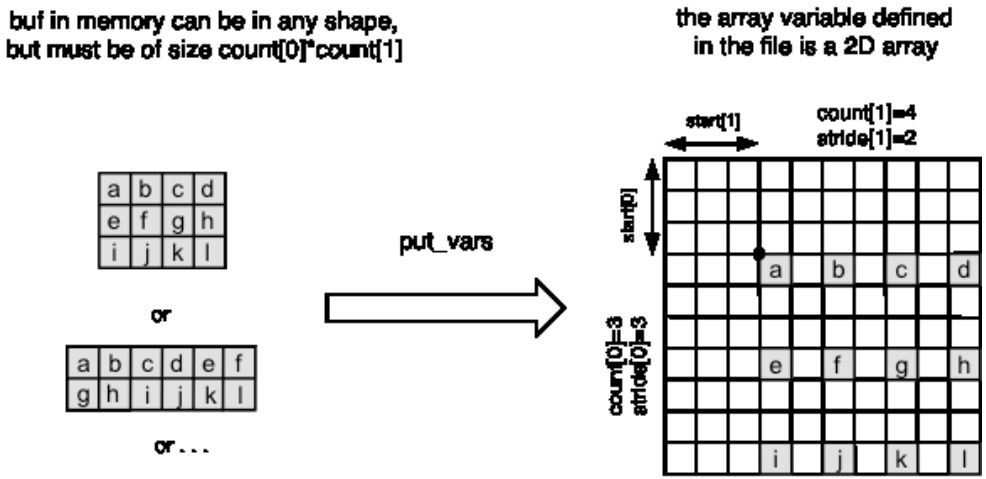
- Also buffered interface is available (`bget`/`bput`) which allow setting the internal buffer

C

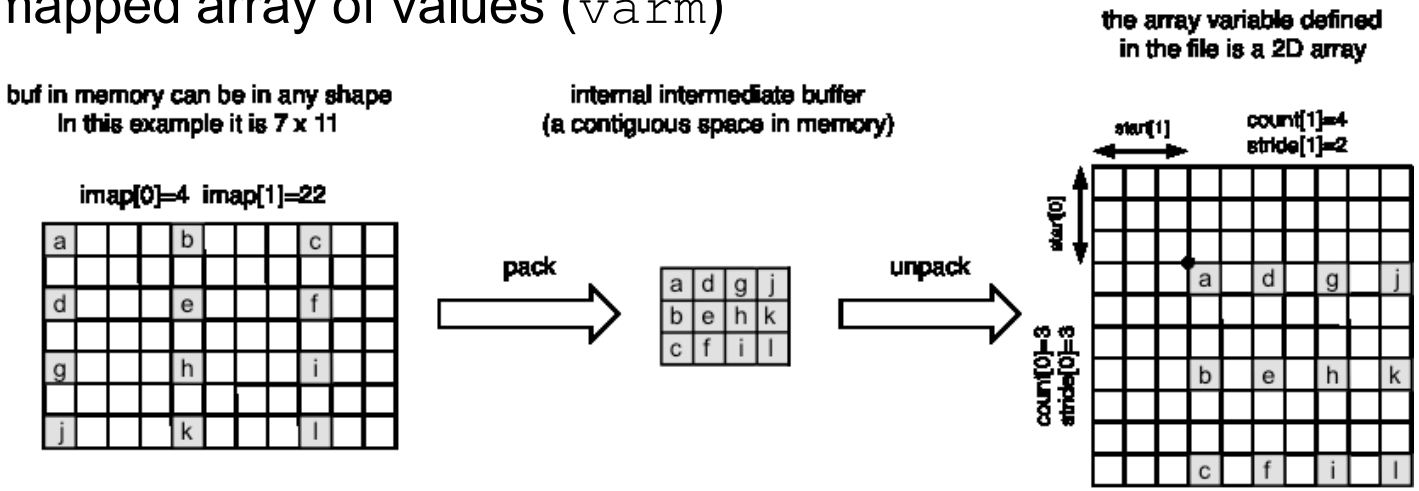
```
int ncmpi_buffer_attach(int ncid, int bufsize)
int ncmpi_buffer_detach(int ncid)
```

Additional access types

- subsampled array of values (`vars`)



- mapped array of values (`varm`)



source: PnetCDF C Interface Guide, http://cucis.ece.northwestern.edu/projects/PnetCDF/doc/pnetcdf-c/

API matrix

`ncmpi_...` `nfmpi_...`	blocking	non-blocking	buffered

single data value	`put_var1`
	`get_var1`
		`iput_var1`
		`iget_var1`
			`bput_var1`
			`bget_var1`
			(since 1.3.0)

array of values	`put_vara`
	`get_vara`
		`iput_vara`
		`iget_vara`
			`bput_vara`
			`bget_vara`
			(since 1.3.0)

subsampled array of values	`put_vars`
	`get_vars`
		`iput_vars`
		`iget_vars`
			`bput_vars`
			`bget_vars`
			(since 1.3.0)

mapped array of values	`put_varm`
	`get_varm`
		`iput_varm`
		`iget_varm`
			`bput_varm`
			`bget_varm`
			(since 1.3.0)

list of subarrays of values	`put_varn`
	`get_varn`
	(since 1.4.0)
		`iput_varn`
		`iget_varn`
		(since 1.6.0)
			`bput_varn`
			`bget_varn`
			(since 1.6.0)

Version updates of parallel-netcdf

- New in version 1.3.1:
- PnetCDF now duplicates the MPI communicator internally
- New datatypes `NC_UBYTE`, `NC_USHORT`, `NC_UINT`, `NC_INT64`, `NC_UINT64`, and `NF_INT64` and bigger arrays using `(NC/NF)_64BIT_DATA` (CDF-5)
- New C APIs: `ncmpi_put_vara_ushort`, `..._uint`, `..._longlong`, and `..._ulonglong`. Similarly for var1, var, vars and varm APIs.
- New Fortran APIs: `nfmpi_put_vars_int8` and similarly for var1, var, vars, varm

Version updates of parallel-netcdf

- New in version 1.4.1:
- Fortran API syntax changes in `nfmpi_put_att` and `nf90mpi_put_att` family (e.g. Intent(IN), ...)
- Introduction of **Subfiling**:
  Divides a file transparently into several smaller subfiles; master file contains all metadata about array partitioning information among the subfiles; transparently for user

C

```
MPI_Info_create(&info)
MPI_Info_set(info, "nc_num_subfiles", "4")
ncmpi_create(…, info, fh)
```

- New in version 1.5.0 of parallel-netcdf: C++ API
- New in version 1.6.0: Conform with netCDF4 on CDF-1 and CDF-2 formats

Outline

- Introduction
- Basic file handling
- Advanced file operations
- Exercises

Exercise

Exercise 3 – 1d array non blocking

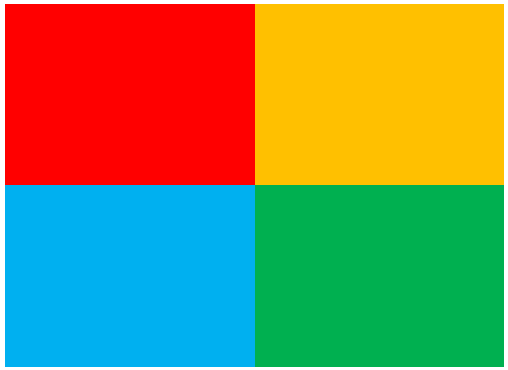
Modify the write-program of Exercise 2 as follows:

- Instead of writing all 10000 values using a single write statement, write 10 times 1000 elements
- Use a non-blocking write routine
- Use your existing read-program to validate your file

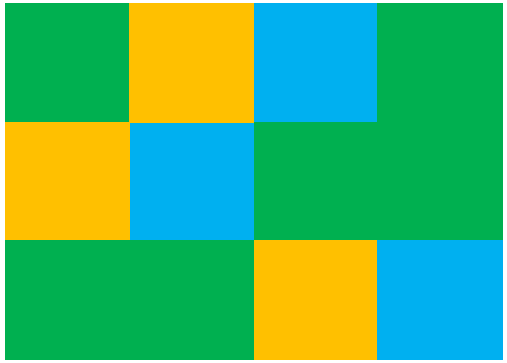
Exercise

stride

static



master-worker, workers write



master-worker, master writes

Exercise

Exercise 4 – Mandelbrot set

- Implement a solution for the stride decomposition (`type = 0`) of the Mandelbrot example (`mandelpnetcdf.c` or `mandelpnetcdf.f90`)
- Use a collective blocking write call
- You will need the following pnetcdf routines
  - `ncmpi_create / nfmpi_create`
  - `ncmpi_def_dim / nfmpi_def_dim`
  - `ncmpi_put_att_double / nfmpi_put_att_real`
  - `ncmpi_def_var / nfmpi_def_var`
  - `ncmpi_put_vara_int_all / nfmpi_put_vara_int_all`
  - `ncmpi_close / nfmpi_close`

Hints

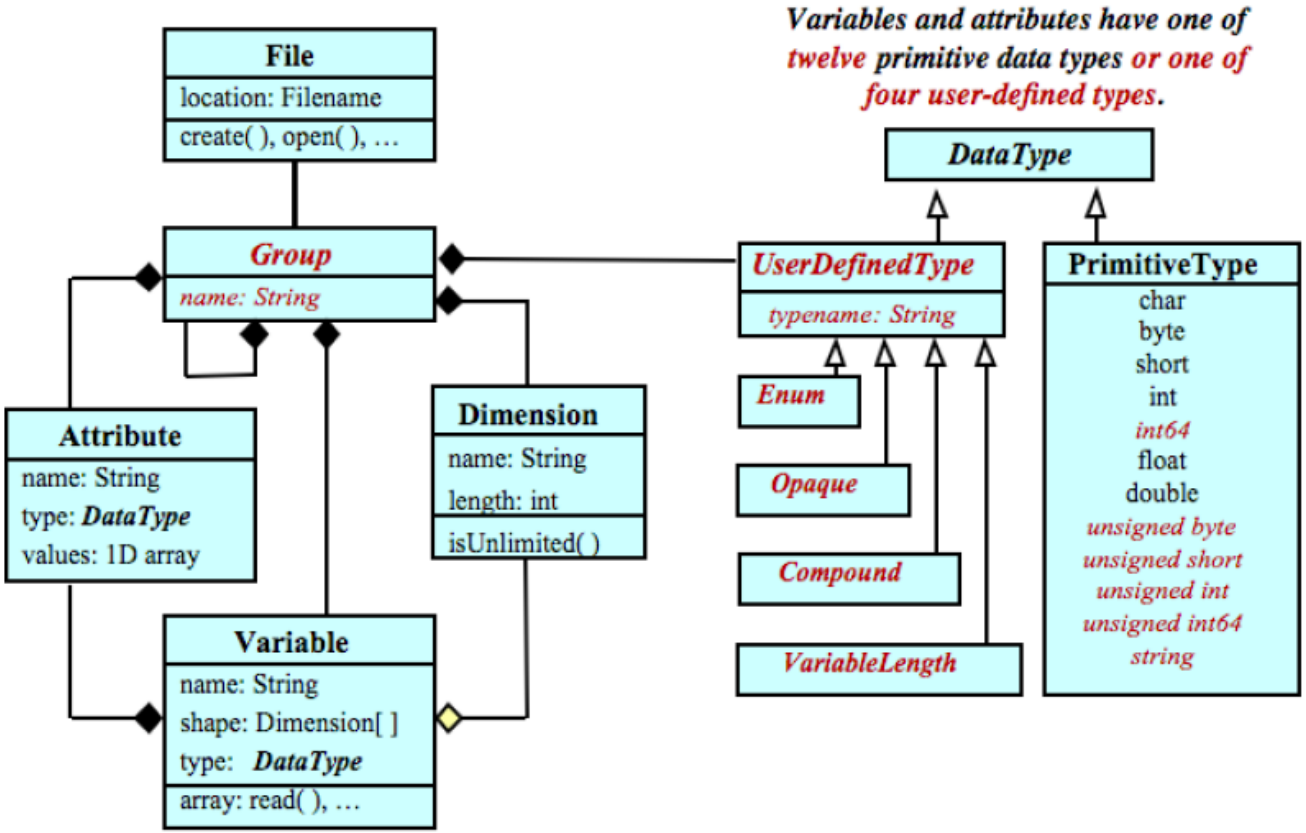
Options for running the program:

`-t 0` (stride decomposition)

`-f 3` (use pnetcdf)

```
/lustre/jwork/hpclab/train000/exercises2015/par_mandel[_fortran]/
```

NetCDF 4 model



A file has a top-level unnamed group. Each group may contain one or more named subgroups, user-defined types, variables, dimensions, and attributes. Variables also have attributes. Variables may share dimensions, indicating a common grid. One or more dimensions may be of unlimited length.

source: Hartnett, E., 2010-09: NetCDF and HDF5 - HDF5 Workshop 2010.